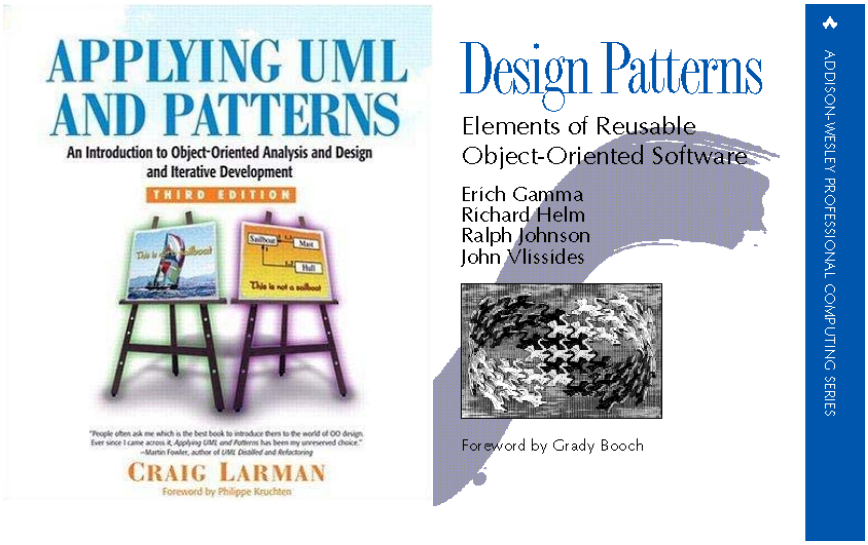
Object-Oriented Analysis and Design using UML and Patterns

Course Overview



Goal of the Course

To provide a thorough understanding of OO analysis and design with UML and patterns

To follow the process of OO analysis and design from requirements capture through to design using the Unified Process as the framework

Course Topics

Learn How to Think in Objects!

Fundamental Concepts of OO
Objects, Classes, Inheritance, Polymorphism, etc.

UML and (Agile) Unified Process (UP)

Use Cases and Use Case Model

Object-Oriented Analysis (OOA)
Domain Model, Analysis Model, etc.

Object-Oriented Design (OOD)
Design Model, Use Case Realization, RDD, Heuristics, Patterns, etc.

Architectural Issues
Logical Architecture, Layers, Subsystems, etc.

Advanced OO Principles
Solid Principles

Condition of Satisfaction

You will know you are succeeding when:

You can read and understand UML diagrams

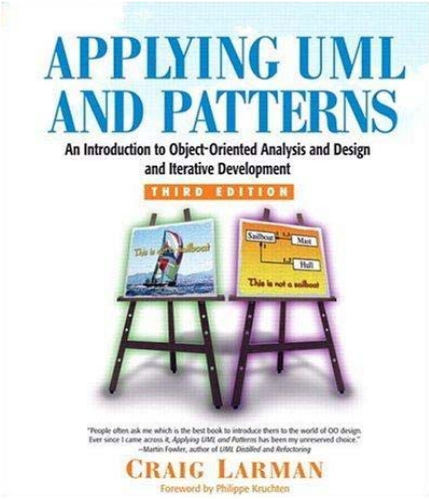
You can produce UML models in the laboratory work

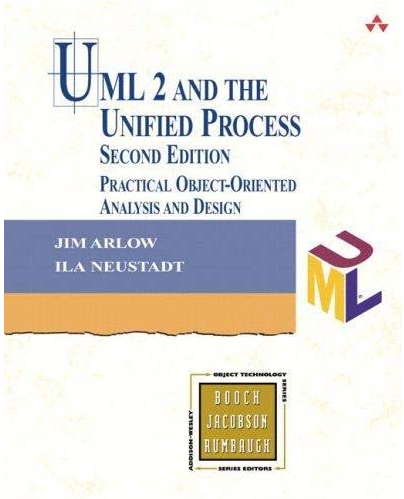
You apply your knowledge effectively back at your workplace

References

"*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*", Craig Larman, Prentice Hall, 3rd ed., 2005

"*UML 2 and The Unified Process: Practical Object-Oriented Analysis and Design*", Jim Arlow & Ila Neustadt, Addison-Wesley, 2nd ed., 2005



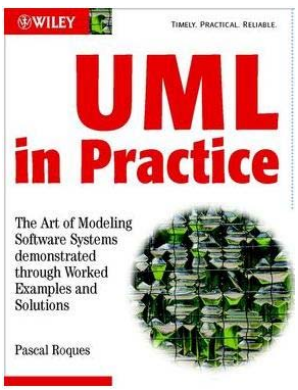


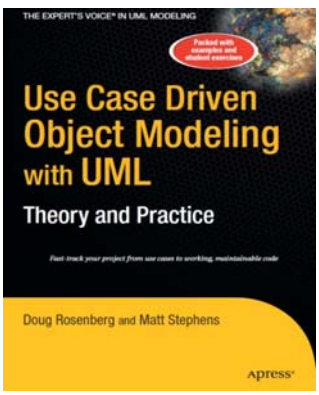
References

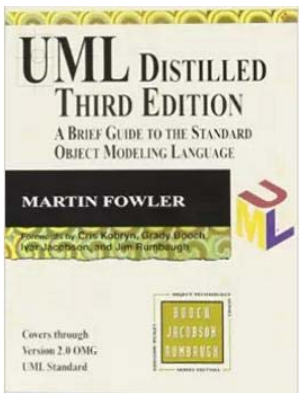
"*UML in Practice: The Art of Modeling Software Systems Demonstrated through Worked Examples and Solutions*", Pascal Roques, John Wiley & Sons Ltd., 2004

"*Use Case Driven Object Modeling with UML: Theory and Practice*", Doug Rosenberg & Matt Stephens, APress, 2008

"*UML Distilled*", 3rd ed., Martin Fowler, Addison-Wesley, 2003



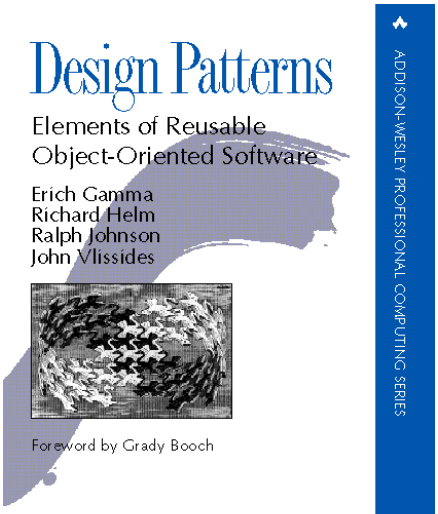




References

***"Design Patterns: Elements of Reusable Object-Oriented Software"*, Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.**

Gang of Four (GoF)



Structured Thinking
vs.
Object-Oriented Thinking

Dealing with Changing Requirements

Let's say that you are an instructor at a conference.

People in your class had another class to attend following yours, but don't know where it is located.

One of your responsibilities is to make sure everyone knows how to get to their next class.

Structured Approach

1. **Get the list of people in the class.**

2. **For each person on the list:**
   ① Find the next class they are taking.
   ② Find the location of that class.
   ③ Find the way to get from your classroom to the person's next class.
   ④ Tell the person how to get to their next class.

Structured Approach: Code Skeleton

```
main() {
   …
   get the list of student
   for (each student in the list) {
      schedule = getSchedule(student);
      nextClass = getNextClass(currentClass, schedule);
      source = findLocation(currentClass);
      destination = findLocation(nextClass);
      showDirection(source, destination);
   }
}

getSchedule(…) { … }
getNextClass(…) { … }
findLocation(…) { … }
showDirection(…) { … }
```


Other Approach (i.e., OO Approach)

You post directions to go from this classroom to the other classrooms and then tell everyone in the class, *"I have posted the locations of the classes following this in the back of the room, as well as the locations of other classrooms. Please use them to go to your next classroom."*

You will expect everyone

- know what their next class is
- can find the classroom they are to go from the list
- can follow the directions for going to the classrooms themselves

What's the difference?

The biggest difference is the

"Shift of Responsibility"

Objects and Responsibilities

Instructor

- Telling people to go to next classroom

Student

- Knowing which classroom they are in
- Knowing which classroom they are to go to next
- Going from one classroom to the next

Classroom

- Having a location

Direction Giver

- Given two classrooms, giving directions from one classroom to the other

OO Approach

1. **Start the control program.**
2. **Instantiate the collection of students in the class.**
3. **Tell the collection to have the students go to their next class.**
4. **The collection tells each students to go to their next class.**
5. **Each student**
   - ① Find where his next class is.
   - ② Determines how to get there.
   - ③ Goes there.
6. **Done.**

OO Approach: Code Skeleton

```
main() {
   …
   for (each student in the list) {  student.goNextClass();   }
}
```

```
class ClassRoom { … }

class DirectionGiver {
   …
   void showDirection(…) {
      find locations for current
     and next class
      …
   }
}
```

```
class Student  {
   …
    void goNextClass() {
       directionGiver.showDirection(
          currentClass, nextClass);
    }
}
```

Object-Oriented Analysis and Design using UML and Patterns

Fundamentals of
Object-Oriented Concepts

Object-Oriented Concepts

Objectives

Define objects, classes, inheritance, and polymorphism

Compare interface inheritance vs. implementation inheritance

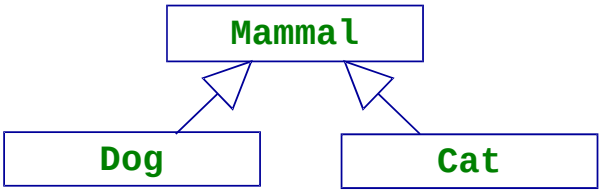
Understand dynamic binding mechanism

OO paradigm builds on three important concepts of objects, classes and inheritance

Objects

Classes

Inheritance



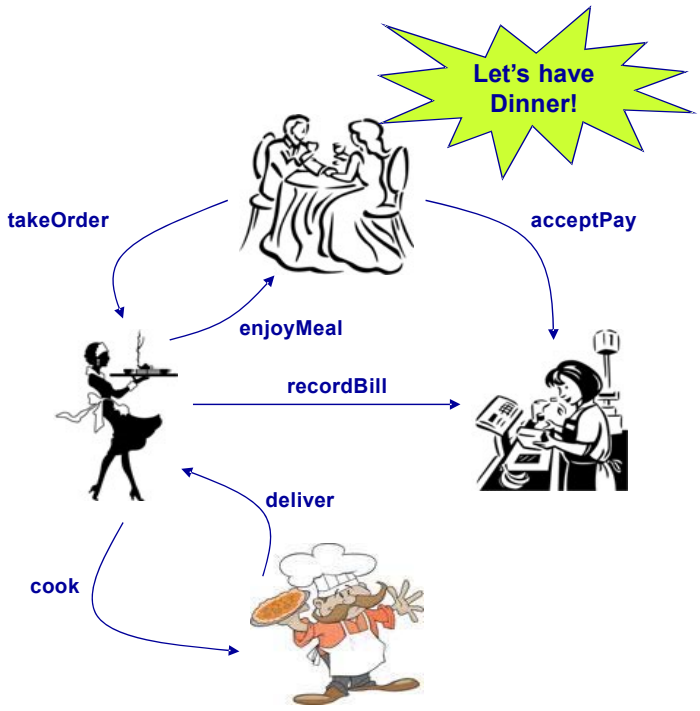
OO also requires 'polymorphism' (i.e., dynamic binding)

Objects are building blocks of software systems

A program is a collection of interacting objects.

Objects communicate by sending 'messages' to each other.

In this way, they cooperate to complete a task.



Objects are things or concepts with crisp boundaries and meaning *(for the problem at hand)*

Tangible Things

- Customer
- Store
- Parking Lot
- Car
- Tire
- …

Concepts & Processes

- Strategy
- Contract
- Class Meeting
- Finding a path through a maze
- Sorting a card deck
- …

Objects must have externally observable behaviors as a result of performing operations

Every object has operations each of which denotes a *service* that an object offers to its clients (or *responsibility* that it performs for them).

The services/responsibilities are represented as a set of operations.

The set of operations constitute an *interface* (or *type*) of an object.

Objects may have attributes that describe them

Are these also objects?

- Color
- Beauty
- Hungry
- Anger
- Love
- Time
- Age
- Address
- …

} **Attributes**

Objects may have states which could affect how objects act or react

Can you get cokes or snacks at any time from the machine? If not, why?

The behavior of the vending machine may depend on time or the order in which one operates on the machine.

This time-dependent behavior implies the existence of state within the vending machine.

The state of an object is determined by the current (usually dynamic) values of each of the attributes

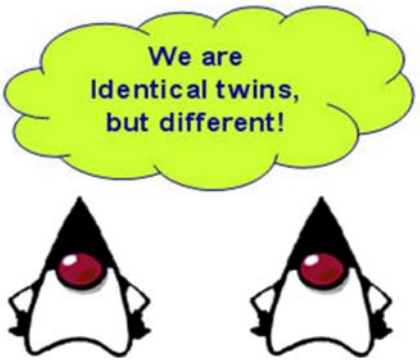
Attributes can be:

- **Coins deposited,**
- **Change,**
- **Height,**
- **Current number for each items**
- **etc**

It is good engineering practice to encapsulate the state of an object rather than expose it.

Objects have unique identities

Identity is that property of an object that distinguishes itself from all other objects.



How many button objects exit in the code?

```
class Button { ... };

Button  b1 = new Button();
Button  b2 = b1;
```

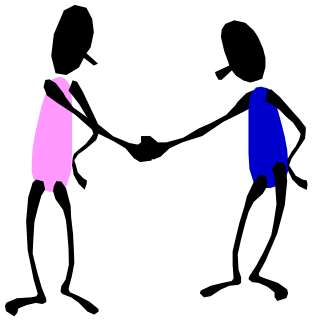
Do not confuse between the name (or handle) of an object and the object itself!

Objects perform actions by making requests of each other through a mechanism of messages

No object exists in isolation. Objects must collaborate with other objects by exchanging *messages.*

A message is a request for action.

An object may accept a message, and in return will perform an action and may return a result.



Message passing is equivalent to *invoking an operation* (*method* in Java or *member function* in C++) on another object.

In a message, there is a designated *receiver* (or *target*) that accepts the message

When we send a message to an object `obj`, the `obj` is a receiver object

```
// In C++, if obj is a pointer var
obj->msg();

// In C++, if obj is a ref. var
obj.msg();

// In Java, obj must be ref. var.
obj.msg();
```

The actual behavior performed by the receiver may be different, depending upon the type of the receiver.

Polymorphism!

Ask not what you can do to your objects, but ask what your objects can do for you

Anthropomorphism
(i.e, live objects)

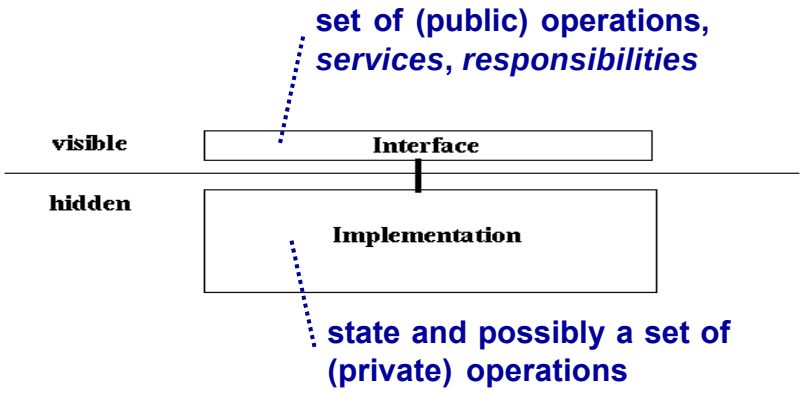
"Instead of a bit-grinding processor ... plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires"
-- Dan Ingalls

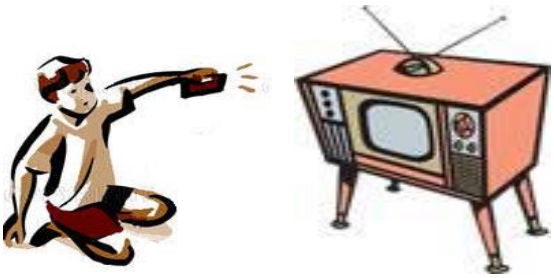
It is good engineering practice to encapsulate implementation details rather than expose it

Encapsulation:
separation of interface from implementation

Object is a black box. Its internal workings and parts are hidden.

Helps code reuse and reliability
- strong cohesion
- loose coupling





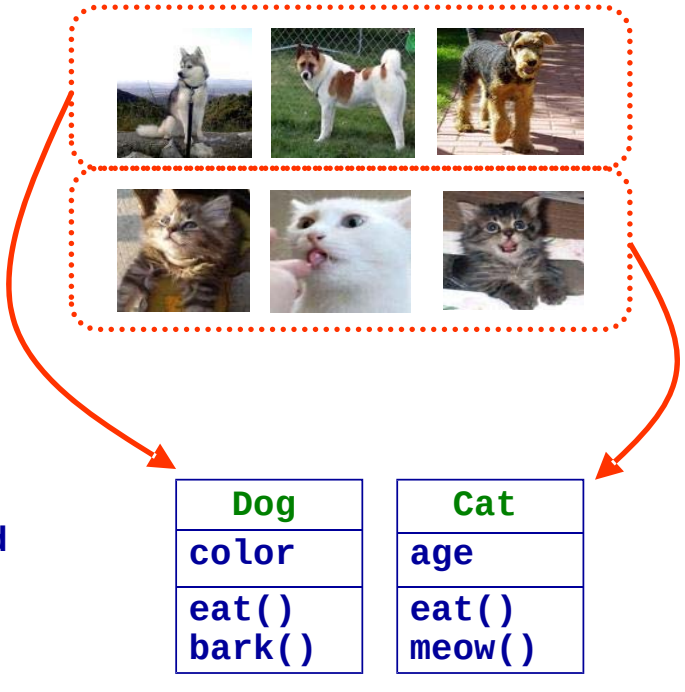
Do not need to know how an object will perform actions, but need to know what messages it will understand

A class is a group of objects with the same attributes, behavior, relationships, and semantics

There are too many objects in the real world, which is impractical to handle

Classifying objects factors out commonality among sets of similar objects
- describe what is common just once
- create any # of copies later

A class is a repository for behavior and the internal representation of the associated objects

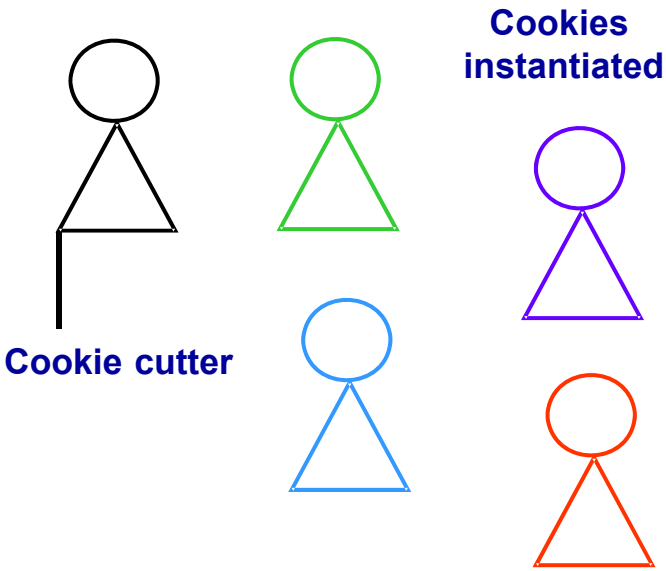


Objects are created from classes through the process of instantiation

Objects are actual *instances* of a class

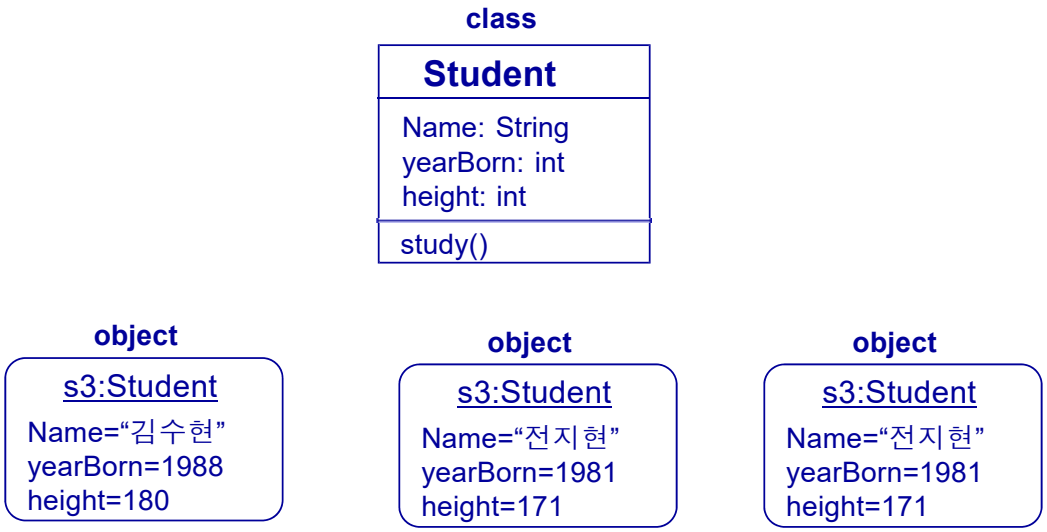
Classes are cookie cutters that make cookies
- **As cookie cutter is not a cookie by itself, a class is not an object by itself.**

Classes are factories and objects are products from the factory.



The terms *instance* and *object* are interchangeable.

Instances created from the same class may or may not have the identical attribute values, but they are unique at all times



Student s1 knows how to do everything student s2 does, but s1 is not s2 or vice versa. Even the s2 and s3 are different.

An object's *interface* is a set of all signatures that can be sent to the object

A operation's *signature* is
type of its return value,
the operation's name,
types of its parameters

```
⟹ void
⟹ paintIcon(
⟹  Component c,Graphics g,int x,int y
)
```

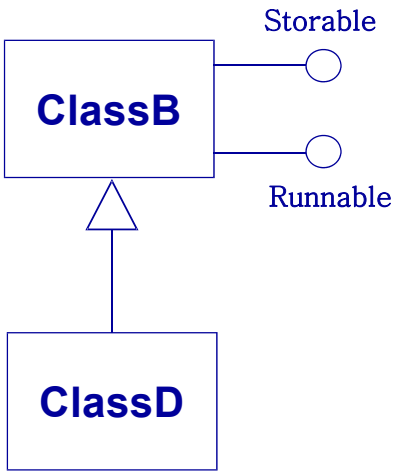
Icon Interface

```
int getIconHeight()
int getIconWidth()
void paintIcon(Component c,Graphics g,int x,int y)
```

An object's *type* is a name denoting only a particular interface

Since a class defines the operations an object can perform, it also defines the object's type.

An object can have many types and objects of different classes can have the same type



Types of a:ClassD

ClassD, ClassB, Storable, Runnable

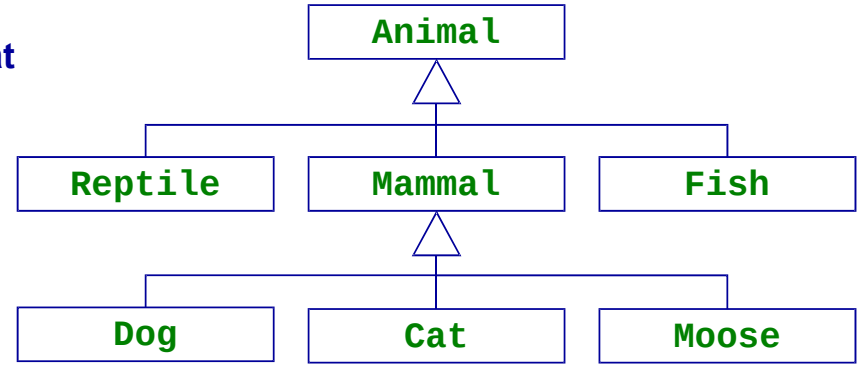
Both a:ClassD and b:ClassX are of type

Runnable

Classes might be organized into a tree called an inheritance hierarchy

Inheritance allows us to arrange class definitions that reflect natural category hierarchies

Information (data and/or behavior) that can be found at a level in a class hierarchy is automatically applicable to lower levels of the hierarchy.



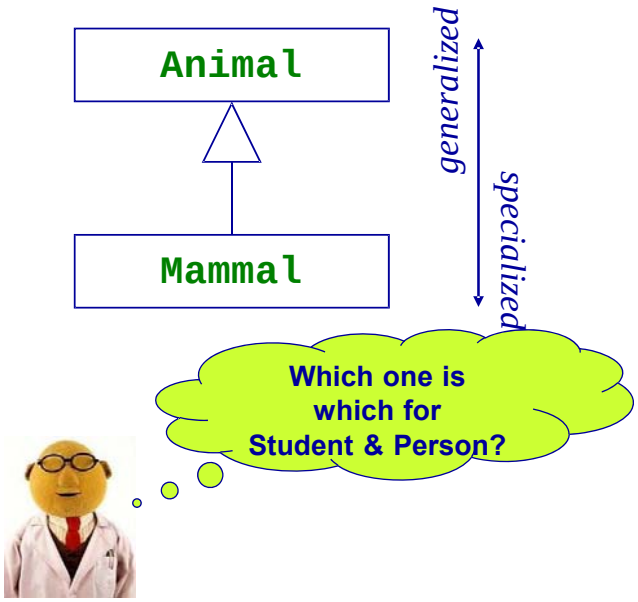
The single most important rule in OO is this: (Public) inheritance means "*is-a*" relationship (aka, *generalization/specialization* relationship)

A Mammal *is an* Animal

A Student *is a* Person

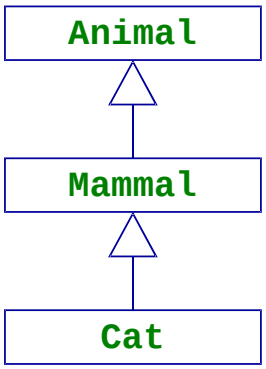
Animal is called a *superclass* or *base class*

Mammal is called a *subclass* or *derived class*



A class can be a superclass as well as a subclass at the same time and the “is_a” relationship is transitive

A Mammal *is an* Animal

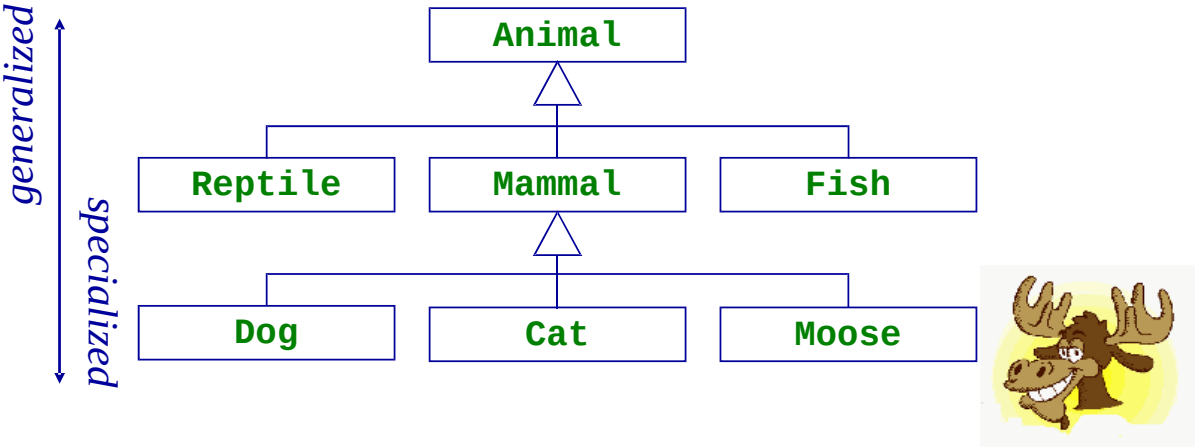


Mammal is a superclass of Cat and a subclass of Animal

A Cat *is a* Mammal

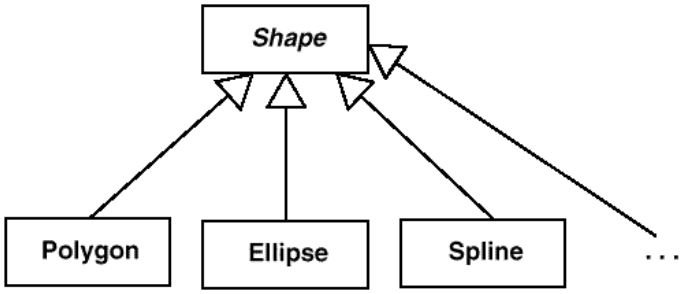
Transitivity: A Cat is an Animal, too

More than one classes can inherit from a given class

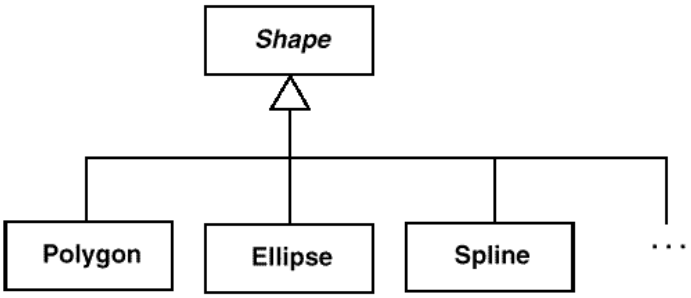


The root provides more general properties shared by all its descendents while the descendents typically add specializing properties which make them distinct among their siblings and their sibling’s descendents

UML Notation

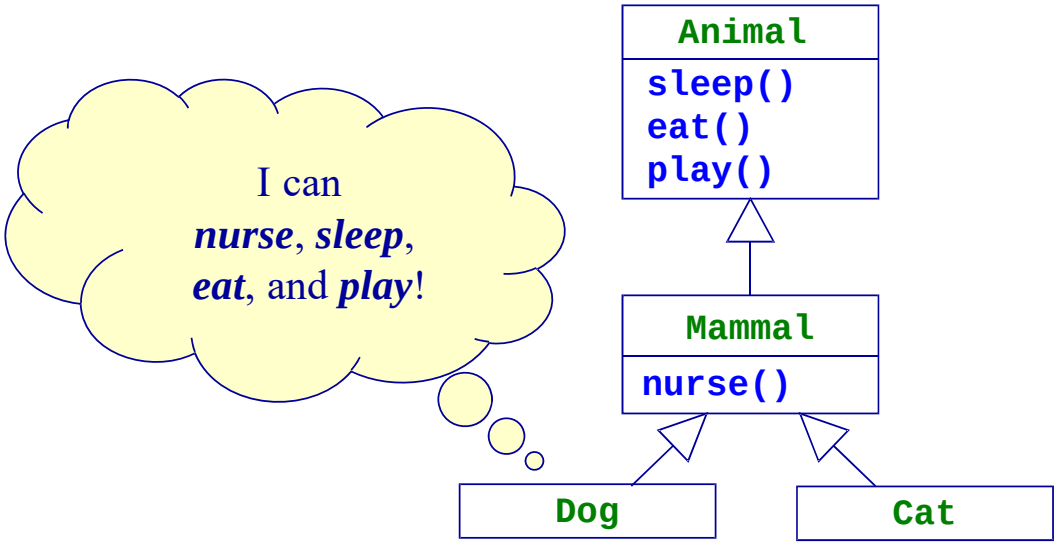


(a) Separate Target Style



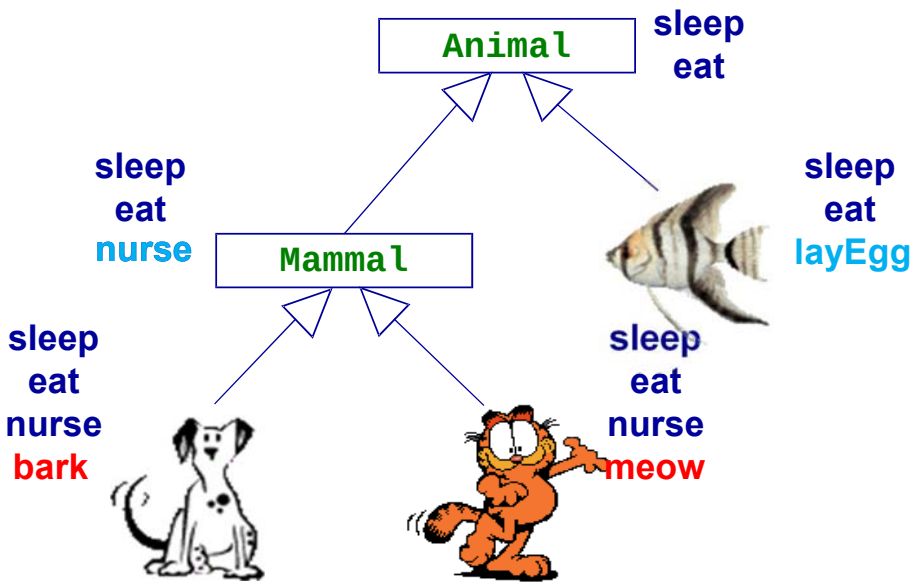
(b) Shared Target Style

Subclasses inherit all attributes and operations from its superclass and its ancestors



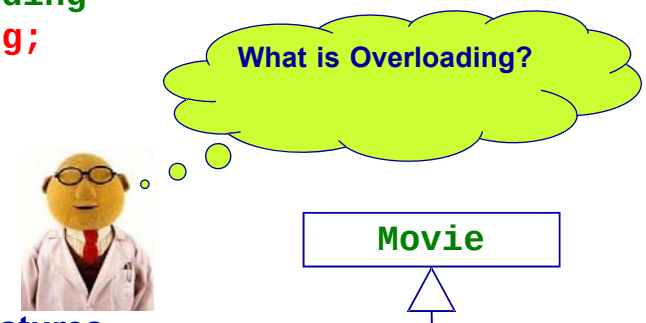
Information (data and/or behavior) that can be found at a level in a class hierarchy is automatically applicable to lower levels of the hierarchy.

Superclass factors out members common to its subclasses



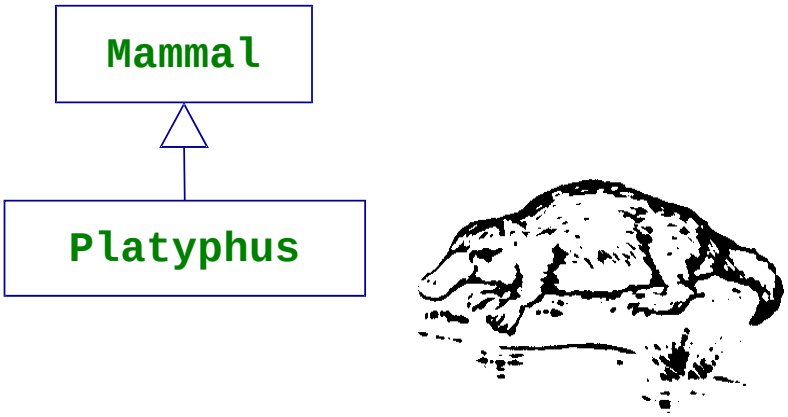
Subclasses can redefine inherited operations (Overriding)

```
public class Movie extends Attraction {
  public int rating() {
    return scripting + acting + directing;
  }
}
public class JamesBondMovie extends Movie {
  public int rating() { // overriding
    return 10 + acting + directing;
  }
}
```



Operations must have the same signatures

One Big Mistake (?) in One of the Famous OO Books!



Subclasses can alter or override information inherited from parent classes:

- ➔ **All mammals give birth to live young.**
- ➔ **A platypus is an egg-laying mammal.**

The practical meaning of the *"is a"* relationship

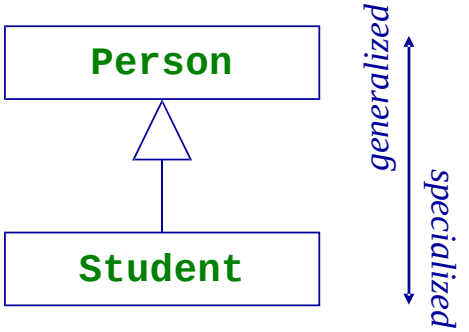
Every object of Student type is also an object of Person type, *but not vice-versa*

Person represents a more general concept than Student, and Student represents more specialized concept than Person

Anything that is true of an object of Person type is also true of an object of Student type, *but not vice-versa*

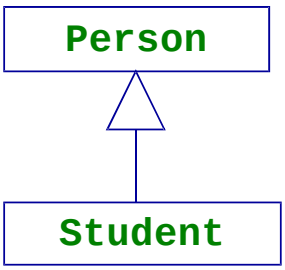
Student *is a* Person

```
class Person { ... };
class Student :
      public Person { ... }
```



Anything a superclass can do, a subclass can do as well

Anywhere an object of Person type can be used, an object of Student type can also be used just as well, *but not vice versa*
→ *Liskov Substitution Principle (LSP)*




```
class Person { ... };
class Student : public Person { ... }

void sleep(const Person& p);      // anyone can sleep
void study(const Student& s);     // only students study (?)

Person p;     // p is a Person
Student s;    // s is a Student

sleep(p);     // OK, p is a Person
sleep(s);     // OK, s is a Student & a Student is-a Person

study(s);     // OK
study(p);     // Error! - p is not a Student
```


A pointer (or reference) to a subclass can be implicitly converted to a pointer (or reference) to a superclass, but *not vice versa*

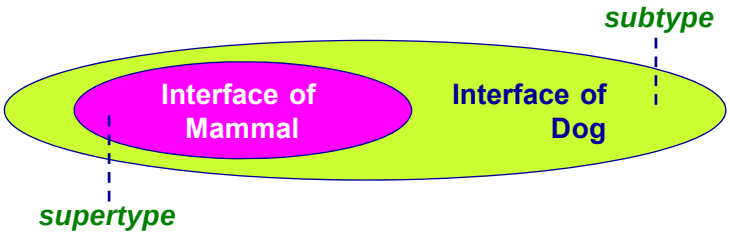
```
Student s;
Person* pPtr = &s;   // OK

Person p;
Student* sPtr = &p;
```

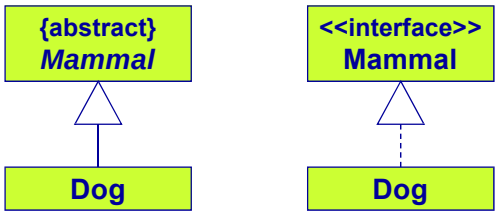
Error! - needs explicit casting
`sPtr = dynamic_cast<Student*>(&p)`

Subclass designers decide what to share: Interface or Implementation?

A type is a *subtype* of another if its interface contains the interface of its *supertype*



Subtyping uses inheritance as a mechanism for *interface sharing*
- enforces the *"is_a"* relationship
- also called *interface inheritance*



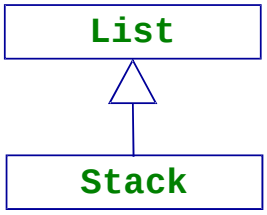
Most common mechanism for interface sharing

Subclass designers decide what to share: Interface or Implementation? (Cont'd)

Subclassing uses inheritance as a mechanism for *implementation sharing* (i.e., code and representation sharing)

- enforces *"implemented in terms of"* relationship

- also called *class inheritance* or *implementation inheritance*

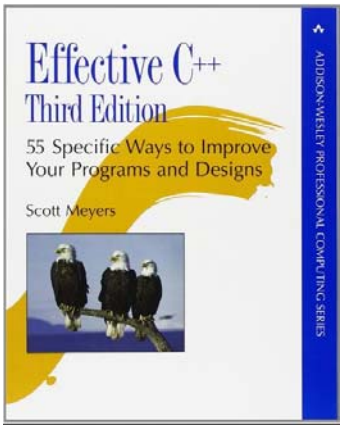


The interface of List should not be visible to the clients of Stack!

Superclass designers must decide what to pass down for each operation when gets inherited.

```
class Shape {
public:
   virtual void draw() const = 0;           // pure virtual
   virtual void error(const string& msg);   // virtual
   int objectID() const();                  // nonvirtual
}

class Circle : public Shape { ... };
class Rectangle : public Shape { ... };
```



Inheritance comes in two flavors at function-level:

- ***inheritance of function interface only* and**

- ***inheritance of function interface & implementation***

As a class designer, we need to explicitly specify our intention using appropriate types of member functions (or methods)

Pure virtual function: `virtual void draw() const = 0;`

To have derived class inherit a *function interface only*

Virtual function: `virtual void error(const string& msg);`

To have derived classes inherit *a function interface as well as a default implementation*

Nonvirtual function: `int objectID() const();`

To have derived classes inherit *a function interface as well as a mandatory implementation*

A reference (or pointer) variable has two types associated with it

Static type

- the type declared at program
- fixed and never changed

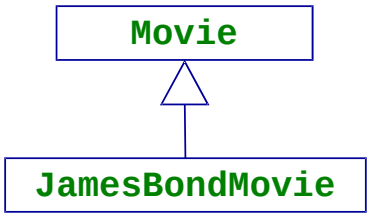
Dynamic type

- the type of object it actually refers to
- can be changed during lifetime

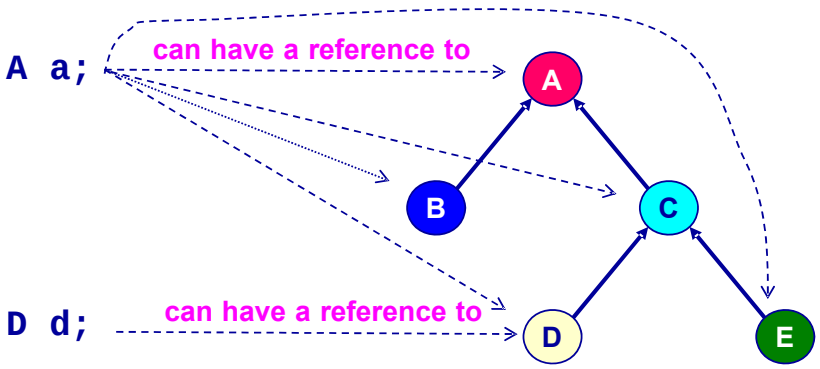
Example:

```
Movie m = new Movie();
m = new JamesBondMovie();

JamesBondMovie jm;
jm = new JamesBondMovie();
m = jm;
jm = m; // error!
```



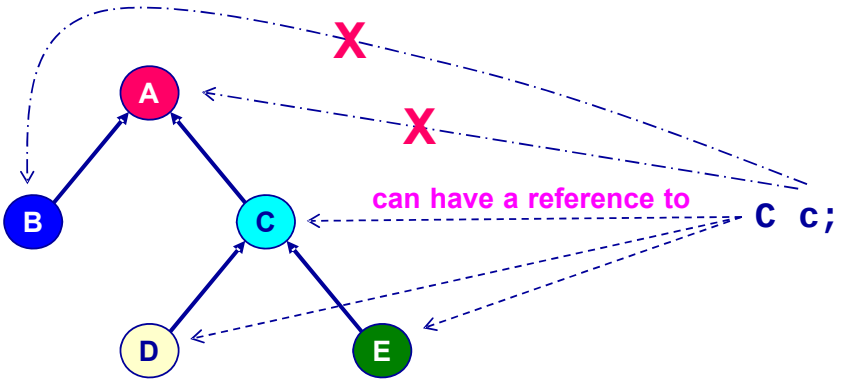
A subclasses can be assigned to its superclass, but not *vice versa*



Rule: A reference variable of type *A* (e.g., `Movie`) can have a reference to an instance of any subclass *extended* from *A* (e.g.,a `JamesBondMovie`)

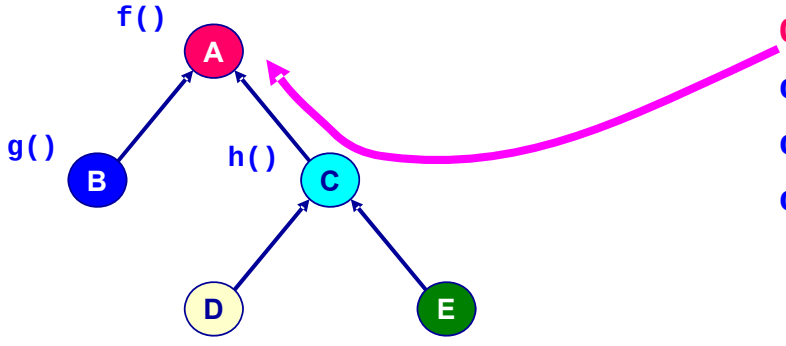
```
A a = new A(); // OK
a = new C();   // OK
B b = new B(); // OK
          …
a = b;         // OK
```

A superclass cannot be assigned to its subclass




```
A a = new A(); // OK
C c = new C(); // OK
a = c;   // OK
…
c = a;   // Error!
```


When a message is sent to a receiver object, the compiler checks its legality by performing an operation lookup using static type




```
C obj = new C();
obj.h();    // OK
obj.f();    // OK
obj.g();    // illegal!
```

The compiler searches up starting from the static type until it finds the invoked operation

A subclass can do anything a superclass can do, but not vice versa

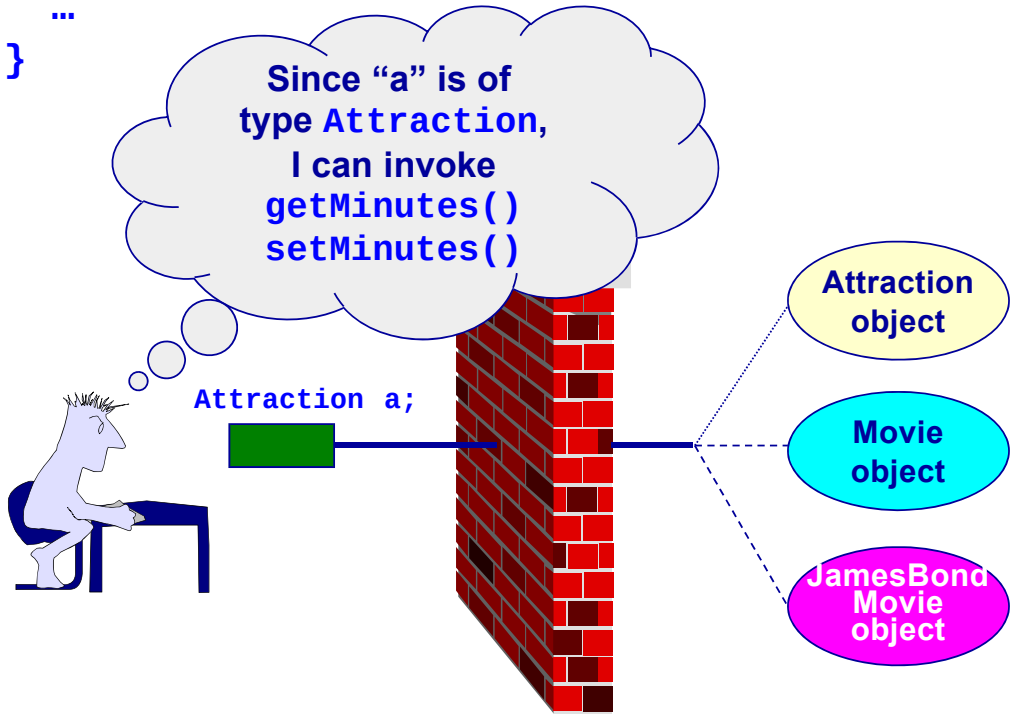
```
void someMethod(Attraction attraction) {
  int minutes = attraction.getMinutes();
  …
  attraction.setMinutes(minutes);
}
```

Since user of an object can access it's operations only through a reference variable, all she knows is the set of operations defined in the class of the variable's static type.

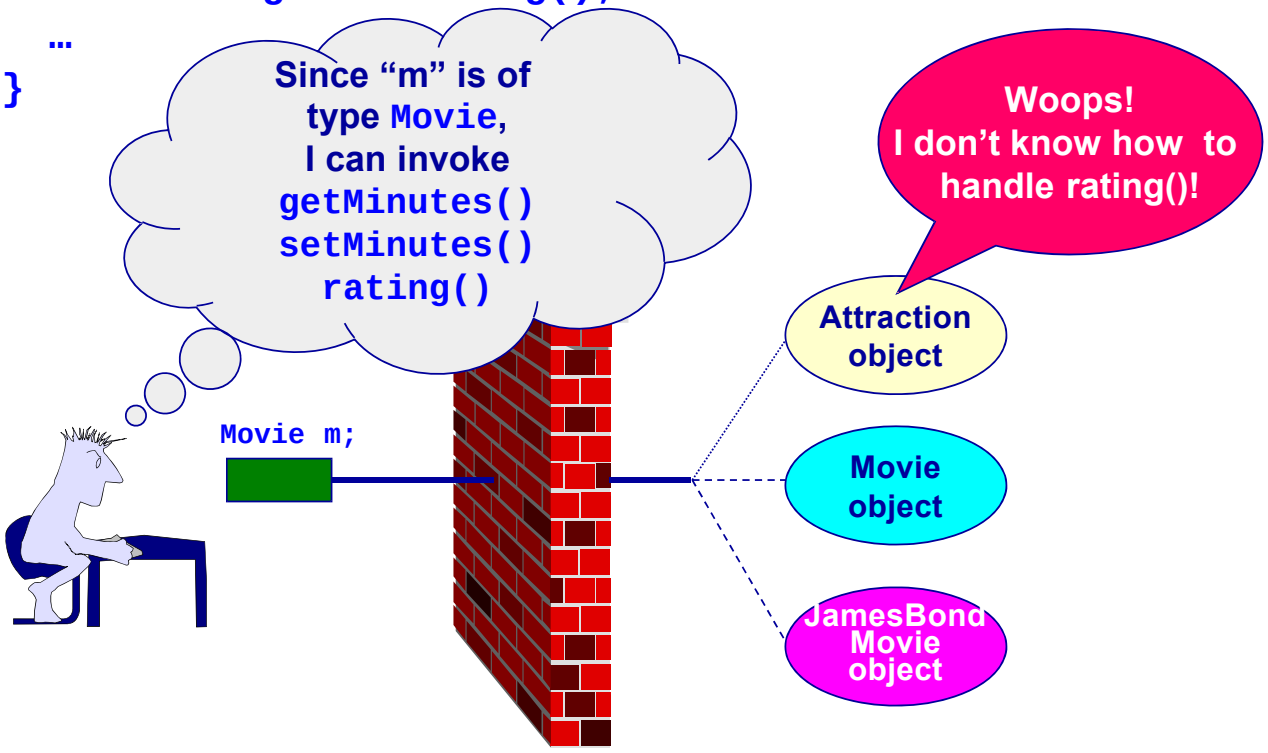
A subclass inherits operations from all of its ancestors

Therefore, it is safe to send a message for its superclass to an instance of a subclass, but *not vice versa*.

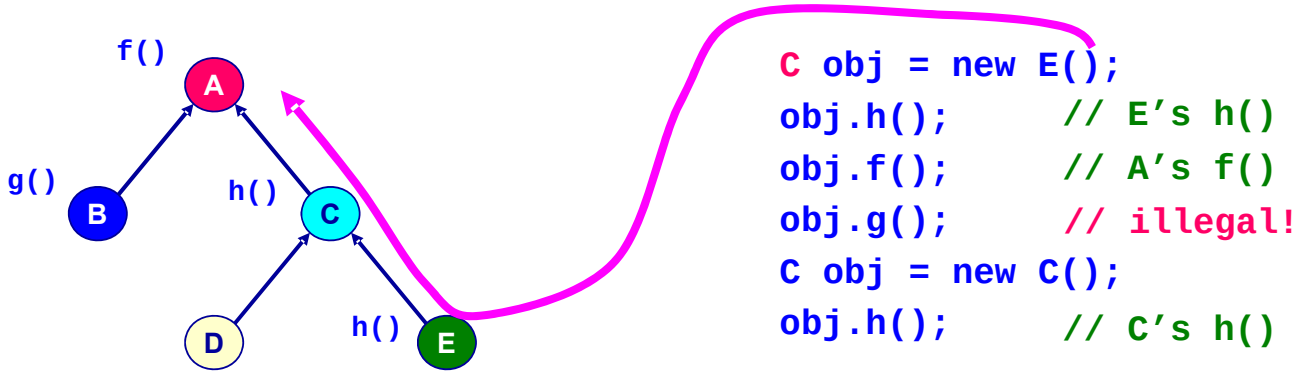

```
void someMethodA(Attraction a) {
  int minutes = a.getMinutes();
  …
}
```



```
void someMethodB(Movie m) {
  int rating = m.rating();
  …
}
```



When a message is sent to a receiver, the runtime selects an operation using dynamic type




```
C obj = new E();
obj.h();      // E’s h()
obj.f();      // A’s f()
obj.g();      // illegal!
C obj = new C();
obj.h();      // C’s h()
```

The runtime searches up starting from the dynamic type until it finds the first invoked operation

Actually, the determination of what behavior to perform may be made at compile-time or at run-time

At compile-time

- static binding or early binding
- static functions in Java/C++
- nonvirtual functions in C++

At run-time

- dynamic binding or late binding
- virtual functions in Java/C++

Even if we send the same message to objects, the behavior can be different depending on the receiver object -- *polymorphism*

Poly (multi) + *Morphism* (form), i.e., a fancy word for multi-forms

When a method must *accept* an instance of superclass as a parameter, it can accept the instance of any subclasses

Likewise, when a method must *return* an instance of superclass, it can return the instance of any subclasses

1. Consider the following declaration:

class B extends A { ... } **// Assume public inheritance used**

What this declaration implies? Select all that apply.

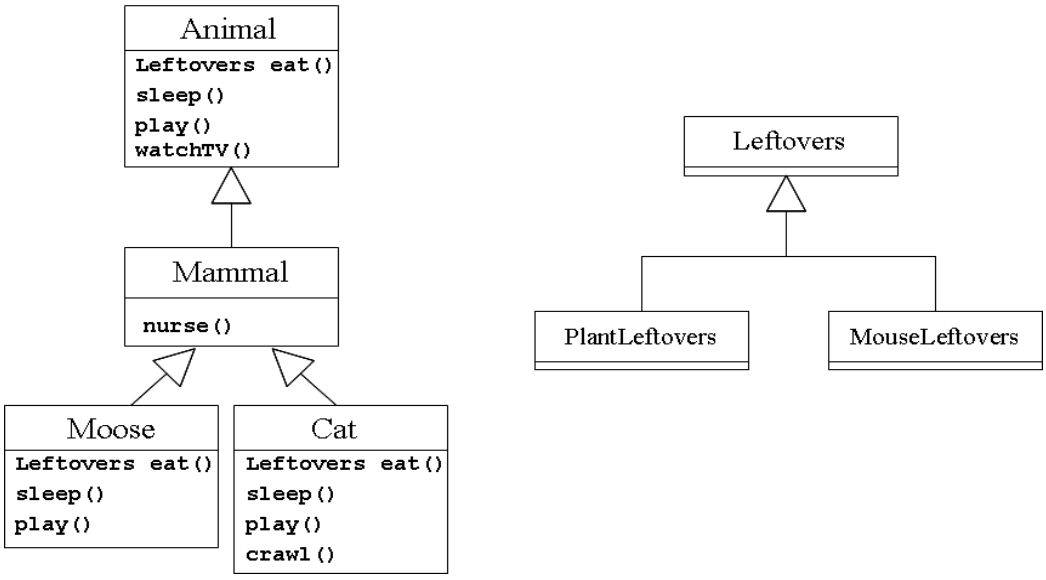
① Every object of type B is also an object of type A, but *not vice-versa*.

② Anything that is true of an object of type B is also true of an object of type A, but *not vice-versa*.

③ A represents a more general concept than B, and B represents a more specialized concept than A.

④ Anywhere an object of type B can be used, an object of type A can be used as well, but *not vice-versa*.

Consider the following class diagrams. All the operations are public.



1. List all operations that objects of **Mammal** can perform.

2. List all operations that objects of **Moose** can perform.

3. List the name(s) of the class(es) that *Cat::eat()* can return as a result.

※ For each of the statement below, mark "***Yes***" or "***No***" depending on the legality of the statement. If "***Yes***", indicate which method will be executed. For example, if the method to be invoked is in the **Cat** class, denote it as "*Cat::sleep()*".

4.
```
Animal anim = new Moose();
anim.play();
```
(a)_____
(b)_____

5.
```
Mammal mamm = new Cat();
mamm.eat();
mamm.crawl();
```
(a)_____
(b)_____
(c)_____

6.
```
PlantLeftovers r = new Leftovers();
```
(a)_____

7.
```
Animal anim;
Moose myMoose = new Moose();
anim = myMoose;
anim.eat();
myMoose = anim;
```
(a)_____
(b)_____
(c)_____

8.
```
Cat myCat = new Cat();
MouseLeftover rem = myCat.eat();
```
(a)_____

※ Consider the following method ***foo*** and answer either "***Yes***" or "***No***"

```
void foo(Mammal m) { ... }
```

9. Is it legal to call *m.watchTV()* in *foo*? (a)_____
Is it legal to call *m.crawl()* in *foo*? (b)_____

Object-Oriented Analysis and Design using UML and Patterns

Motivation for Object Technologies

Motivation for Object Technologies

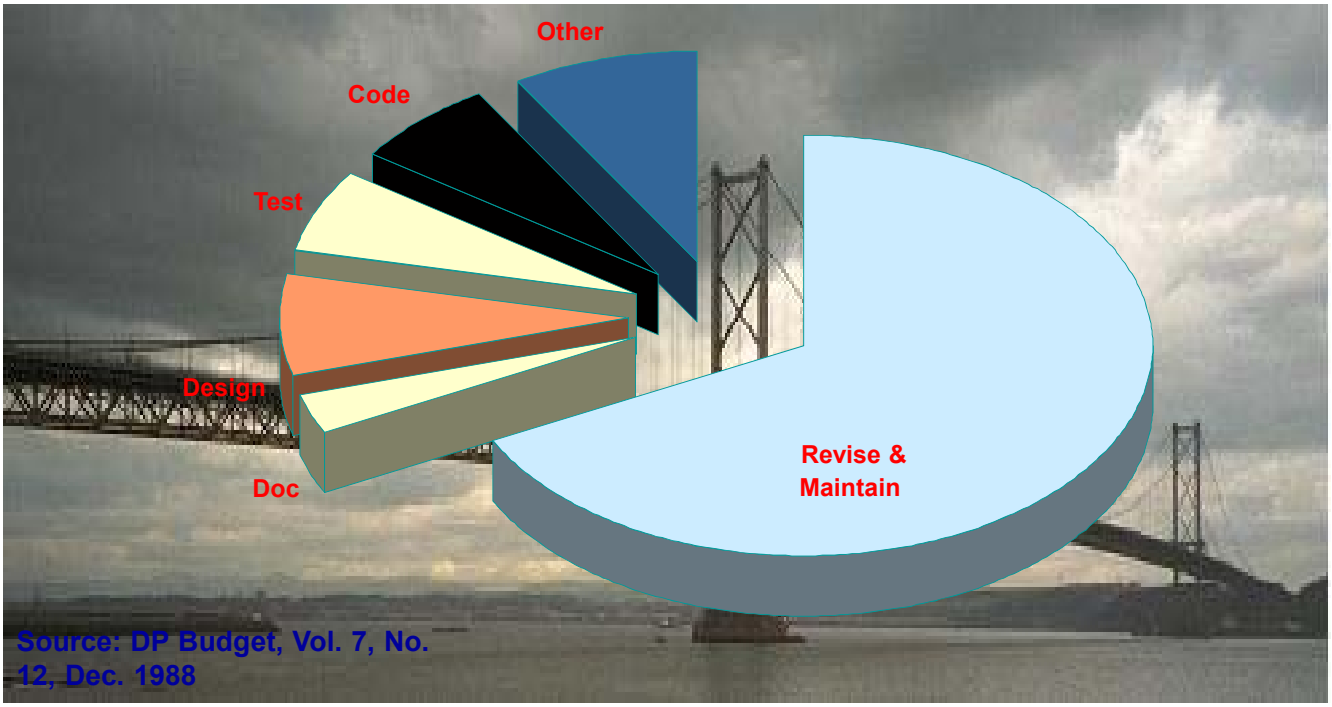
Objectives

Provide motivation for the object-oriented technology

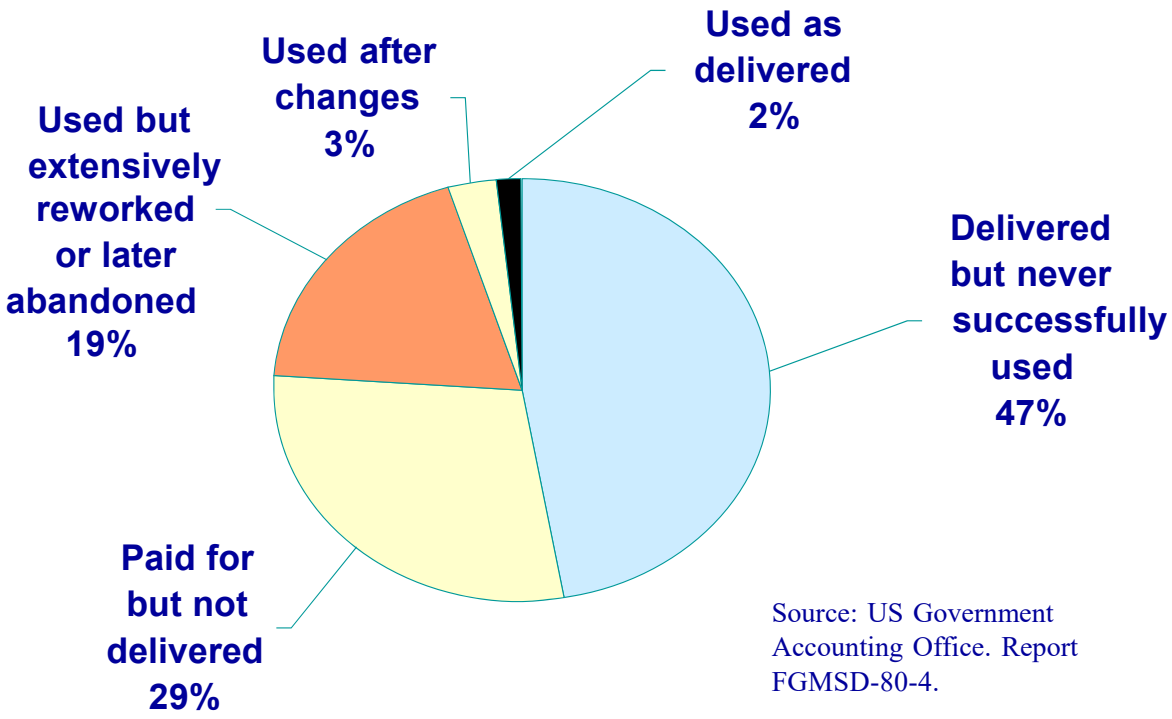
Introduce design patterns and frameworks

Compare structured vs. OO approach in problem solving

Strategic rational system development plans are based on the complete cost of a system, not solely on development costs



Randomly selected US government software projects:
An AT&T study indicates that, on average, business rules change at the rate of 8% per month



Source: US Government Accounting Office. Report FGMSD-80-4.

The real power and advantage of OT is *its capacity to tackle complex systems* and *to support easily adaptable systems*, lowering the cost and time of change

***The Corporate Use of OT*, Dec 1997, Cutter Group.**
Prioritized reasons for adopting OT:

1. Ability to take advantage of new operating systems and tools
2. Elegantly tackle complexity & create easy adaptability
3. Cost savings
4. Development of revenue-producing applications
5. Encapsulation of existing applications
6. Improved interfaces
7. Increased productivity
8. Participation in "the future of computing"
9. Proof of ability to do OO development
10. Quick development of strategic applications
11. Software reuse

To obtain flexible and reusable systems, it is better to base the structure of software on the objects rather than on the actions

Rationale behind OO paradigm:

In general, systems evolve, functionality changes, but data objects, interfaces, and components relations tend to remain relatively stable over time.

Use it for large systems &
for systems that change often

Any other benefits?

It is essential to decompose the complex software system into smaller and smaller parts, each of which may then refine independently (i.e., *stepwise refinement*)

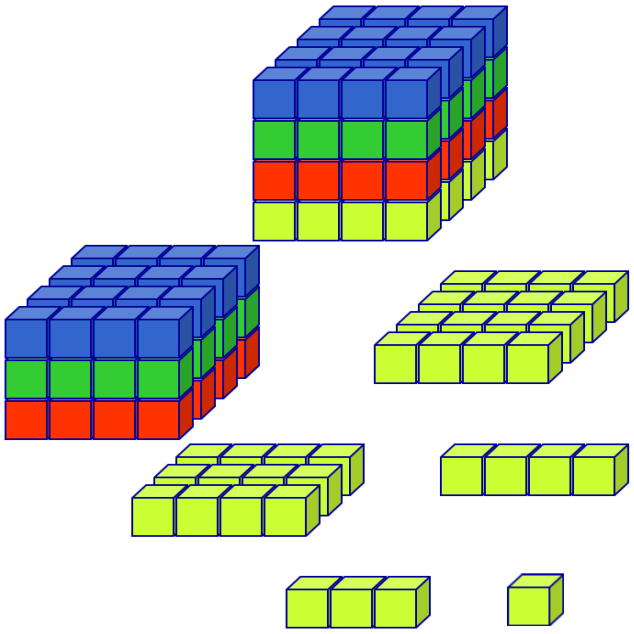
Maximum number of chunks of information an individual can simultaneous comprehend is the order of 7 ± 2

--- Miller (1956)

The technique of mastering complexity has been known since ancient times:

Divide and Conquer

-- Dijkstra (1979)



Structured vs. Object-Oriented Decompositions

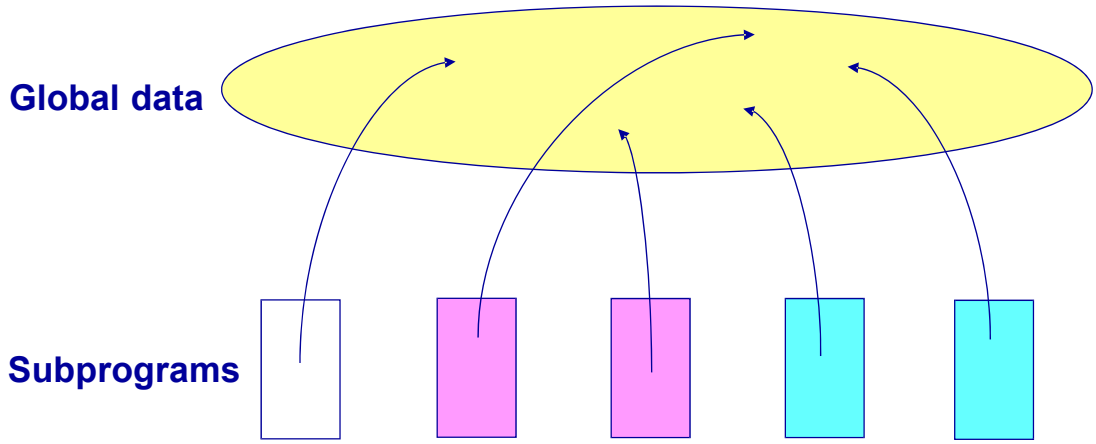
Structured Decomposition

- **Organize a system around procedures/functions**
- ***Program = (Algorithms + Data Structures)***
- **SA/SD/SP**

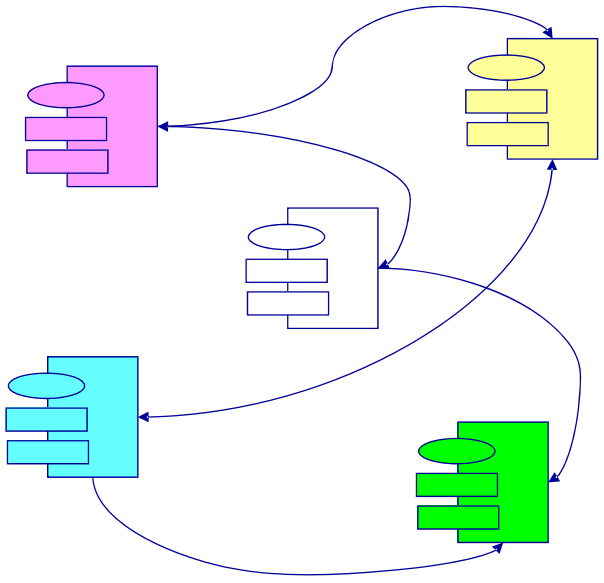
Object-Oriented Decomposition

- **Organize a system around objects**
- ***Object = (Algorithm + Data Structures)***
- ***Program = (Object + Object + ...)***
- **OOA/OOD/OOP**

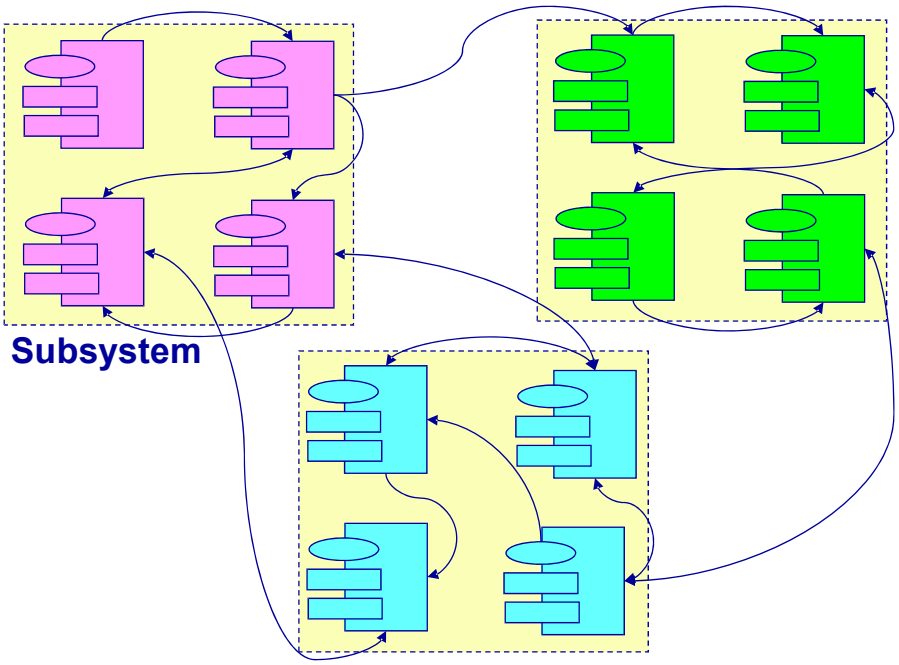
Design Structure Based on Subprograms



Design Structure Based on Objects
(Small Scale)



Design Structure Based on Objects
(Large Scale)



Reuse is not usually achieved or worthwhile at the object-level

Research shows no relationship between increased reuse and collecting a library of reusable components from prior projects.

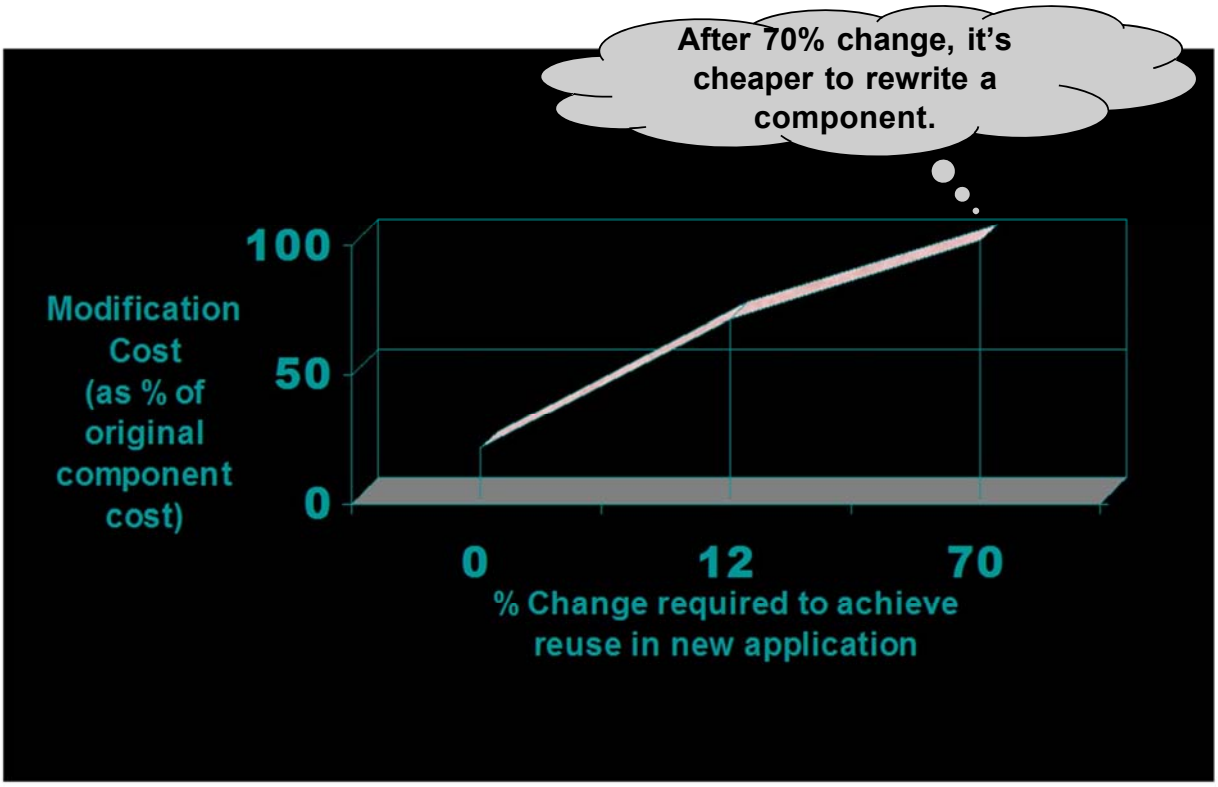
-- Communications of the ACM, pp 75-87 June, 1995

Focus on:

A culture of *framework* creation and use.

Reuse of architecture, analysis and *design patterns*

Reuse at What Level?



Design pattern is a description of a *problem*/*solution* pair in a certain *context*

> **"Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same time twice."**
>
> ***-- Christopher Alexander***

Design Pattern Example: Adapter Pattern

Name: ***Adapter***

Also known as: **Wrapper**

Context: **Client objects call methods of a Supplier object**

Problem: **Client objects expect another interface than the Supplier provides**

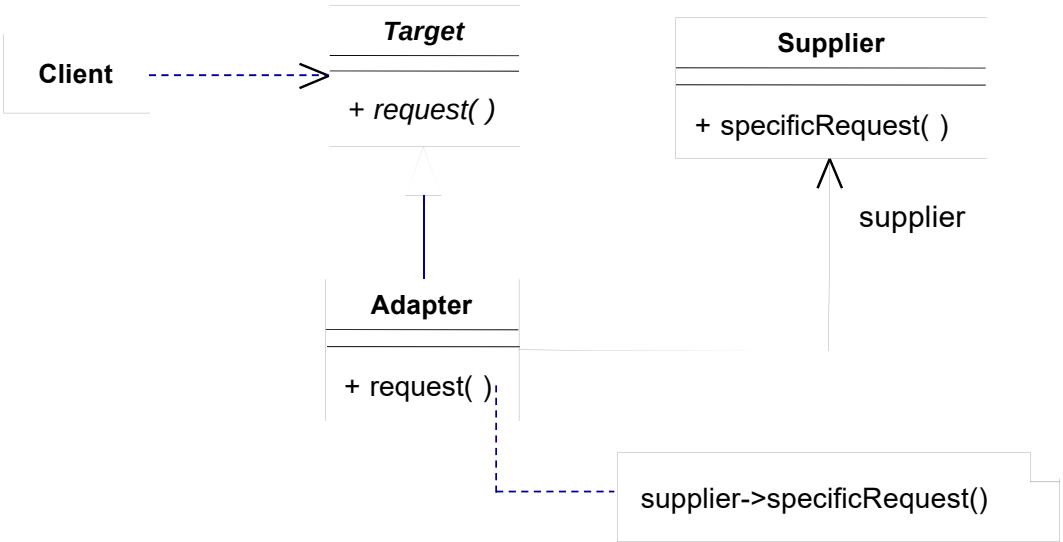
Solution: **Use an Adapter object which adapts one interface to the other**

Solution alternatives:

Class adapter
(relies on multiple inheritance like C++ or Eiffel, or interface inheritance like Objective C or Java)

Object adapter
(relies on single inheritance and delegation)

Object Adapter Solution

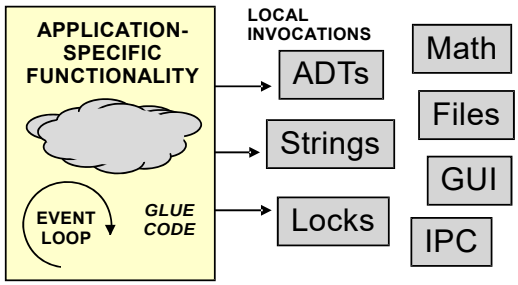


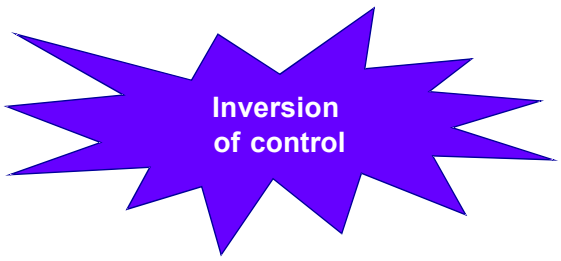
A framework is a class library, but more than an ordinary toolkit (such as math, file i/o, data structures ...)

An integrated set of cooperating classes	**A *semi-complete application*: abstract framework classes are specialized in the application**
Inversion of control	**The "main event loop" is often in the framework, rather than in the application code** **Code in the framework can invoke code in the application by dynamic binding**
Domain-specific	**business, telecommunications, windows system, databases, etc.**

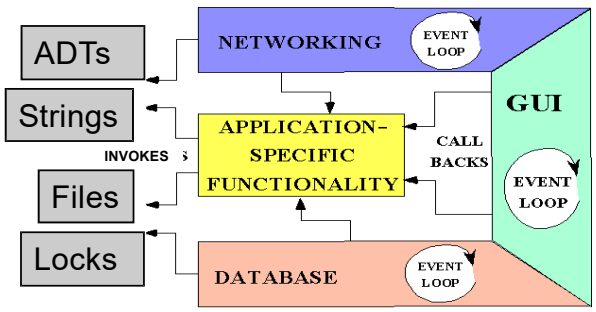
Hollywood Principle: Don't call me, we'll call you!

Class Library Architecture





Framework Architecture

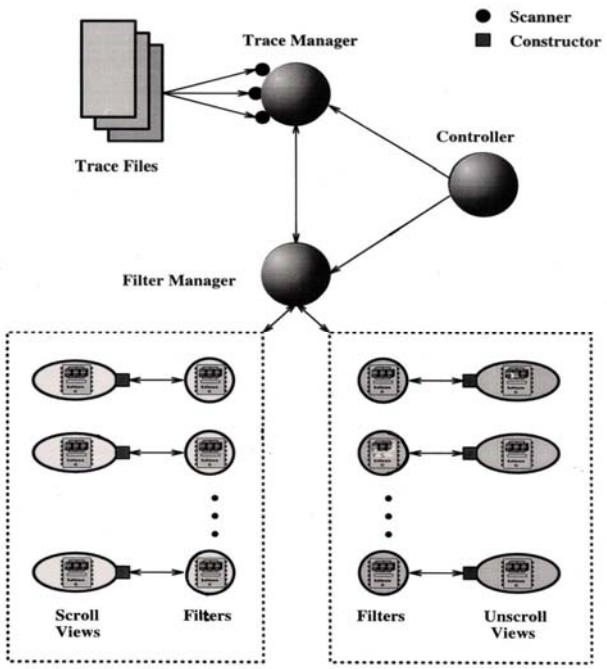


Frameworks allow us to reuse design and code

Framework: family of similar applications

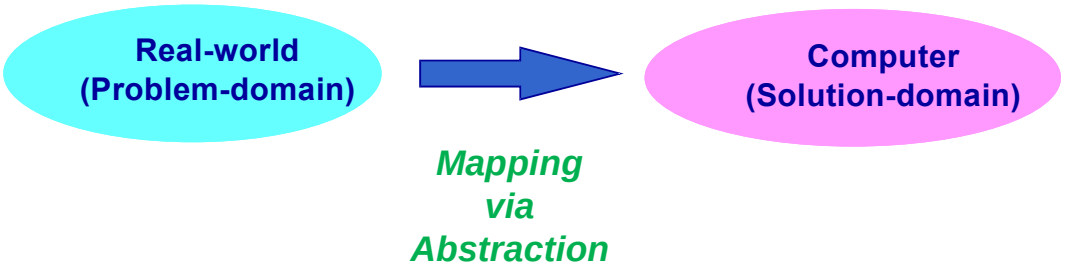
Very fast application development

Powerful parameterization mechanism (subclassing and dynamic binding)



Evolutionary Approach
vs.
Revolutionary Approach

Object-oriented paradigm is an *evolutionary* approach to mastering the software complexity



Assembly Language (Mnemonics)
Human-oriented Languages (High Level Languages)
Subprograms (Procedures or Functions)
Modules
Abstract Data Types
Objects

Subprograms

Subprograms were not fully appreciated as abstractions originally. Instead, they were seen as labor-saving devices.

Libraries of subprograms (such as mathematical or input/output libraries) provided the first hints of *information hiding*.

They permit the programmer to think about operations in high level terms, concentrating on *what* is being done, not *how* it is being performed.

But they are not entirely effective mechanism of information hiding.

Problems with Subprograms

```
int dataStack[100];
int dataTop = 0;

void init() { dataTop = 0; }
void push (int val)
{ dataStack[dataTop++] = val; }
int  pop()
{ return dataStack[--dataTop]; }
int  top()
{ return dataStack[dataTop - 1]; }
```


Modules

Modules basically provide encapsulated collections of subprograms and data with import and export mechanisms.

Solves the problem of encapsulation.

- **Separation of interface from implementation**

Maximum cohesion within a module and minimum coupling among modules are most desirable.

But what if your programming task requires two stacks?

Abstract Data Types (ADTs)

Programmer-defined data type which involves *a set of data values* together with *a set of operations* that can be performed on the values of that data type.

Exports a type definition.

Protects the data associated with the type so that they can be operated on only by the provided operations.

Allows to make multiple instances of the type.

Objects are a natural extension of ADTs, i.e.,
"ADT+*inheritance* "==> innovations in code sharing and reusability.

Objects: ADTs with Inheritance, and Polymorphism

Characteristics of Objects:

- **Encapsulation**
  - **similar to modules**
- **Instantiation**
  - **similar to ADT's**
- **Inheritance and polymorphism**
  - **different *interpretations* of messages with different objects**
  - **a new form of software reuse using *inheritance* and *polymorphism***